IDA PAPER P-1893

Ada* FOUNDATION TECHNOLOGY

Volume III: Software Requirements for WIS Software Design, Description and Analysis Tools

Leon G. Stucki, *Chairman*
Robert J. Knapper, *IDA Task Force Manager*
Elizabeth K. Bailey
Leon J. Osterweil
Christine Youngblut
William E. Riddle, *Past Chairman*

With Additional Contributions by
Grady Booch

John Salasin, *Program Manager*

December 1986

Prepared for

WIS Joint Program Office
Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

IDA Log No. HQ 86-30820

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and sound analytical methodology and that the conclusions stem from the methodology.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

AD-A178434

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) P-1893 - Volume III	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION WIS Joint Program Management Office 8b OFFICE SYMBOL (If applicable) WIS/JPMO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) 7798 Old Springfield Road McLean, VA 22102	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-W5-206 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) Ada™ Foundation Technology: Volume III - Software Requirements for WIS Software Design, Description and Analysis Tools	
12 PERSONAL AUTHOR(S) L. Stucki, R. Knapper, E. Bailey, L. Osterweil, C. Youngblut, W. Riddle, G. Booch	
13a TYPE OF REPORT Final 13b TIME COVERED FROM _____ TO _____ 14 DATE OF REPORT (Year, Month, Day) 1987 January 15 PAGE COUNT 50	
16 SUPPLEMENTARY NOTATION	

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
World Wide Military Command and Control System (WWMCCS), WWMCCS Information System (WIS), automatic data processing (ADP), Ada programming language, CAIS, design tools, Ada Design Language (ADL), reusable software, test and evaluation tools, metric analysis, project management

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Software Design, Description, and Analysis Tools are tools that will assist in the design, construction, and maintenance of WIS Ada software. This category of tools will include: object manager, Ada PDL, Ada PDL tools, categorization scheme tool, reusable components library system, test and evaluation tools, software metrics analyzer, and a set of management tools.

This volume is the third of a nine-volume set describing projects which are planned for prototype foundation technologies for WIS using the Ada programming language. The other volumes include command language; text processing; database management system; planning and optimization tools; graphics; operating systems; and network protocols.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ■ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1.0 INTRODUCTION

1.1 Purpose

The World Wide Military Command and Control System (WWMCCS) is an arrangement of personnel, equipment (including automatic data processing (ADP) equipment and software), communications, facilities, and procedures employed in planning, directing, coordinating, and controlling the operational activities of U.S. Military forces.

In order to support these high level objectives, the WIS system software must provide an efficient, extensible and reliable base upon which to build this functionality. To develop such system software, several projects are planned for prototype foundation technologies for WIS using the programming language Ada. The purpose for developing these prototypes is to produce software components that:

a. Demonstrate the functionality required by WIS.

b. Use the programming language Ada to provide maximum possible portability, reliability, and maintainability consistent with efficient operation.

c. Are consistent with current and "in-progress" software standards.

Foundation areas in which prototypes will be developed include:

a. Command Languages

b. Software Design, Description, and Analysis Tools

c. Text Processing

d. Database Tools Management System (DBMS)

e. Operating Systems

f. Planning and Optimization Tools

g. Graphics

h. Network Protocols

This report describes general requirements for the Software Design, Description, and Analysis Tools.

1.2 Scope

Software Design, Description, and Analysis Tools are tools that will assist in the design, construction, and maintenance of WIS Ada Software. This broad category of tools will include:

a. Object Manager: A system to store strongly typed, persistent software objects (e.g., source code, parse trees) and to regulate the application of the tools to create and maintain them (Section 2.0).

b. Ada PDL: A Program Design Language (PDL) for use in designing and implementing Ada software (Section 3.0).

c. Ada PDL Tools: A set of tools, such as an Ada syntax-directed editor, formatter/pretty-printer, etc., to be used in assisting the design and implementation, and maintenance of Ada software (Section 4.0).

d. Categorization Scheme Tool: An automated tool to accurately categorize the function, behavior, and other characteristics of reusable components (Section 5.0).

e. Reusable Components Library System: A system to store and track reusable components (Sections 6.0, 7.0).

f. Test and Evaluation Tools: A set of tools to be used for automated testing and evaluation of Ada software (Section 8.0).

g. Software Metrics Analyzer: A tool to collect and display various metrics on Ada software (Section 9.0).

h. Management Tools: A set of tools to be used in assisting the planning and resource management of Ada software projects (Section 10.0).

This document will present requirements specifications for these tools and for the associated studies and schema that some tools will require.

1.3 Terms and Abbreviations

ACVC	Ada Compiler Validation Capability
ADL	Ada Design Language
ADP	Automatic Data Processing
AJPO	Ada Joint Program Office
ANNA	ANNotated Ada
APSE	Ada Programming Support Environment
C3	Command, Control, and Communications
CAIS	Common APSE Interface Set
COCOMO	COnstructive COst MOdel
DDA	Design Description and Analyses
E & V	Evaluation & Validation
JCS	Joint Chiefs of Staff
NOSC	Naval Ocean Systems Center
PC	Personal Computer
PDL	Program Design Language
STARS	Software Technology for Adaptable, Reliable Systems
WIS	WWMCCS Information System
WWMCCS	World Wide Military Command and Control System

1.4 References

[Ada 83] U.S. Department of Defense, *The Ada Programming Language Reference Manual*, U.S. Department of Defense, January 1983.

[Ada 84] U.S. Department of Defense, *Ada Portability Guidelines*, Document Reference 3285-2-208/1, U.S. Air Force Systems Command, Electronic Systems Division, Hanscom AFB, September 1984.

[BAIL 85] Bailey, E.K. and J.F. Kramer, "A Framework for Evaluating the Usability of Ada Programming Support Environments," Manuscript submitted for publication, 1985.

[BOEH 81] Boehm, B.W., *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice Hall, 1981.

[BOWE 83] Bowen, T.P., J.V. Post, J. Tai, P.E. Presson, R.L. Schmidt, "Software Quality Measurement for Distributed Systems," Guidebook for Software Quality Measurement, Volume II, RADC-TR-83-175, July 1983.

[BOOC 83] Booch, Grady, *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., 1983.

[BUHR 84] Buhr, R.J.A., *An Ada-Inspired Graphical Design Notation for Manual and Computer-Aided Design of Modular, Concurrent Systems (with Examples)*, Report SCE-84-1, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, January 1984.

[CARG] Cargill, T.A., "The Blit Debugger (A Preliminary Draft)", Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging, 1983.

[CLEM] Clemm, G. and L.J. Osterweil, *The Integration of Toolpack/IST* , Technical Report available from the Department of Computer Science, University of Colorado, 1985.

[CLEM 84] Clemm, G.M., "Odin - An Extensible Software Environment Report and User's Manual", University of Colorado, CS Dept. Tech. Report, 1984.

[COUT 85] Coutaz, J., "Abstractions for User Interface Design," *Computer*, 18, 9 (September 1985): 21-34.

[DePR 83] DePree, Robert W., Pattern Recognition in Software Engineering, *Computer*, May 1983

[DOD 2167] U.S. Department of Defense, TBD.

[FIKE 85] Fikes, Richard, and Tom Kehler, "The Role of Frame-Based Representation in Reasoning," *Communications of the ACM*, 28, 9 (September 1985).

[FTO 81] Feiber, J., Taylor, R.N., Osterweil, L.J., "Newton - A Dynamic Program Analysis Tool Capabilities Specification," U. Colorado Tech. Repon, 1981.

[GANN 85] Gannon, J.D., E.E. Katz, and V.R. Basili, *Metrics for Characterizing Ada Packages*, Draft Technical Report, Department of Computer Science, University of Maryland, 1985.

[GENE 85] Genesereth, Michael R. and Matthew L. Ginsberg, "Logic Programming," *Communications of the ACM*, 28, 9 (September 1985).

[HANS] Hanson, W.J., "User Engineering Principles for Interactive Systems", Fall Joint Computer Conference 19, 1971.

[HART 84] Hartson, H.R. and D.H. Johnson, "Dialogue Management: New Concepts in Human-Computer Interface Development," manuscript submitted to *ACM Computing Surveys* (revised version), March 1984.

[HAYE 83] Hayes-Roth, Frederick, Donald A. Waterman and Douglas B. Lenat, *Building Expert Systems*, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1983.

[HAYE 85] Hayes-Roth, Frederick, "Rule-Based Systems," *Communications of the ACM*, 28, 9 (September 1985).

[HENR 81] Henry, S. and D. Kafura, "Software Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. SE-7, 5 (1981): 510-518.

[KELL 85] Keller, S.E. and J.A. Perkins, "An Ada Measurement and Analysis Tool," *Proceedings of the Annual National Conference on Ada Technology 1985* , pp. 188-196.

[KIEF] Kiefhaber, S.H., "An Implementation and Evaluation of a Screen Debugger for FORTRAN Programs", Master's Thesis, U. Colorado, Dept. CS Tech. Report, 1983.

[LAN 83] Lancaster, J.N., "Naming in a Programming Support Environment," MIT Report LCS/TR-312, August 1983.

[LUCK 84a] Luckham, David C., Friedrich W. von Henke, Bernd Krieg-Brueckner and Olaf Owe, *ANNA: A Language for Annotating Ada Programs*, Technical Report No. 84-261, Program Analysis and Verification Group Report No. 24, Stanford University, July 1984.

[LUCK 84b] Luckham, David C. and Friedrich W, von Henke, *An Overview of ANNA: A Specification Language for Ada* , Technical Report No. 84-265, Program Analysis and Verification Group Report No. 26, Stanford University September 1984.

[McCA 77] McCall, J.A., P. Richards, G. Walters, *Factors in Software Quality*, 3 volumes, RADC-TR-77-369, Rome Air Development Center, 1977.

[NAC 85] Naval Avionics Center, *Survey of Ada-Based PDLs* , Indianapolis, Indiana, January 1985.

[STUC 75] Stucki, L.G. and G.L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," *1975 International Conference on Reliable Software* , sponsored by the ACM, IEEE, and NBS, Los Angeles, 22-24 April, 1975.

[STUC 77] Stucki, L. G., "New Directions in Automated Tools for Improving Software Quality," *Current Trends in Programming Methodology*, Vol 2, edited by R. T. Yeh, Prentice-Hall, Inc., Engelwood Cliffs, New Jersey, 1977.

[STUC 80] Stucki, L. G. and H.D. Walker, "Concepts and Prototypes of ARGUS," *Proceedings of the Symposium on Software Engineering Environments*, Lahnstein, Germany, 16-20 June, 1980; in *Software Engineering Environments*, edited by H. Huenke, North-Holland Publishing Co., Amsterdam - New York.

[TAX 80] *Taxonomy of Computer Science and Engineering*, AFIPS Taxonomy Committee, AFIPS Press, 1980.

[WIS 85] U.S. Department of Defense, *WIS Ada Design Language Standard*, Revised Draft, Electronics System Division, Air Force Systems Command, Hanscom AFB, 25 January 1985.

2.0 OBJECT MANAGER

This chapter specifies an object manager and tool integration mechanism for Ada program development environments.

2.1 Background

The approach for integrating large and diverse collections of software tools is in line with current trends towards attacking increasing software costs and chronically troublesome software quality by transforming software development into a disciplined engineering activity. In creating this new engineering discipline, researchers have increasingly tried to view software as a product and software engineering as a manufacturing process. Researchers now view software as a product which should be built in stages, incrementally tested and analyzed, and both built and maintained through the use of interchangeable, modular parts. Mechanisms and procedures supportive of this process are essential. Thus, there has been a great deal of research into software development methodologies and software tools over the past decade or more.

Work directed toward identification of superior tools and methodologies has been frustrating, however. Researchers have found that the most effective procedures are often most effective simply because they are well supported by tools. Conversely, the most effective tools are often most effective because they have been built to support particular specific methodologies. This realization has served to focus researchers' attention on the need to create large integrated collections of tools capable of supporting whole software methodologies. These tool collections have come to be known as software development environments.

Considerable attention has recently been focused on how to integrate software tools into the environments which are plausible research vehicles and useful development support vehicles as well. Most current software development environments are little more than either large, overloaded tools or untidy collections of poorly coordinated tools. The major responsibility for coordinating the functions of these tools into a coherent manufacturing process is left to users. It has become apparent, however, that the range of tools needed to effectively and adequately support software development as a well-organized manufacturing process is quite large and probably requires more coordination skills from users than it is reasonable to expect. Further, as software methodologies become more complex and sophisticated, the importance of being able to combine and sequence tools in unplanned ways is becoming clear. Often it turns out that some tools are not suitable for some combinations and sequences which are desirable and necessary. These discoveries have now led to a research focus on effective techniques for the integration of tool capabilities in flexible ways.

One of the most promising directions of this research has been to think of an environment as a collection of objects -- namely the objects which compose the software product being manufactured -- rather than as a collection of the tools needed to create and manage the objects and product. This view may be new in software engineering, but is actually quite a traditional view in such applications as banking, insurance and government. In these areas, computer professionals have encouraged users to think of their work as being aimed at the creation and maintenance of large central repositories of information. This has resulted in the creation of large information management systems, most of which

have been quite successful. Clearly there must be considerable merit to thinking of some large production activities as being information centered.

Odin [CLEM 84] can serve as an example of an environment integration system. This object manager views an environment as being a collection of tools centered around an information repository consisting of software development objects. Odin has been successfully used to integrate a collection of mathematical software development tools into a system called Toolpack/IST [CLEM].

2.2 Scope

The development of an Object Manager will entail prototype activities in several directions. First is the specification and development of an effective and efficient object management system.

Second, a distributed version of the Object Manager is needed in order to enable the effective support of software development by teams of workers.

Finally, a specification of a collection of tool fragments which, when integrated by the Object Manager, comprise the basis of a powerful and effective software development environment for Ada is needed.

Each of these activities will now be described in more detail. As an example, in order to facilitate these discussions, a short summary of Odin's capabilities and architecture is presented. More detailed information about Odin can be found in [CLEM].

2.3 Object Manager Architecture

Odin is a command language processor which incorporates an intelligent, optimizing manager for strongly typed software development objects. This language has as its philosophical basis and goal the creation of objects which the user specifies in response to perceived needs for information which those objects contain. Generally, requested objects are created by the action of tools. Thus it may seem that an object-oriented command language is isomorphic to an algorithmic command language. There is an important difference, however, which becomes clear when one considers that it is often quite easy for the user to conceive of and describe an important object which nevertheless may require lengthy and complex tool operations for its creation. For example, it may be easy for a user to conceive of and describe the output from a program test run but it may be quite hard to describe the intricate steps required in order to build the program source text out of a variety of files and libraries, compile, link, load, execute, and then tabulate the results of executing the program. Users should be encouraged to think of these important objects, describe them tersely, and then have the Object Manager supervise and coordinate the tools which build the requested objects as painlessly and noiselessly as possible. The Object Manager should also be responsible for maintaining a persistent store of both requested and other intermediate objects for use in optimizing the servicing of future requests.

Tool use should be encouraged as much as possible, and stumbling blocks to the use of tools should be removed as effectively as possible. The Object Manager must encourage the composition of tool capabilities. It must make it easy to apply one tool to the output of another. From the point of view of the user, this entails thinking of the object

produced by a tool as a derivative of the object or objects which were taken as input to the tool. Thus the product of a tool is an object derived by the tool, and that object can be the basis for further derivations. It is believed that this is quite consistent with the mental models which tool users exploit when managing the development of software. Effective software development requires the creation and use of a variety of views and versions of the program object under development. Some of these views and versions are easily obtained by the action of readily accessible tools (e.g., a compiler, pretty-printer). Some are currently most readily derived by manual or mental processes. Some are very difficult to derive at all because they involve the sort of composition of powerful tools which has just been described. As a result, at present, some of these mental models do not get constructed at all and some are constructed incompletely or incorrectly. Whatever models the user is able to conjure up, are stored and managed in a haphazard, manual, and error-prone way.

The approach to this problem is to conceive of all such models, mental or otherwise, as objects, store them in a central repository, and consider all of them to have been created from each other by the action of a tool or sequences of tools (except for a few "atomic" objects, which have been externally inserted into the information repository). In this way the user is released from the burden of constructing them and maintaining their consistency with each other as the software development process proceeds and necessitates changes.

One major obstacle to user acceptance of the proposed scenario of widespread use and concatenation of tools is the perception that tools are expensive to use. Indeed, tools, especially some of the more powerful analysis tools which can be expected in an Ada environment, can be quite expensive to use. On the other hand experience has shown that often a variety of powerful, high level tools all rely upon identical or similar bodies of lower level information. These bodies of lower level information are viewed as objects which have been derived by low level tools (e.g., parsers and lexical analyzers). Thus, the preferred strategy is to save such low level objects, once they have been created as part of the operation of another high level tool. These so-called persistent objects are ordinarily retained in the expectation that they will be of use as the basis for the operation of another high level tool. In addition to reusing existing objects, the Object Manager is responsible for reclaiming storage space when necessary. This is done by deleting low-priority derived information, with the knowledge that this information can be re-derived. This strategy may be viewed as an optimization process where what is being optimized is the use of persistent objects across the extent of an entire software development activity, constrained by the amount of storage space available. Alternatively it may be viewed as the process of reusing software objects, and of constructing software tools in such a way as to effectively exploit such reusable objects. In this sense we view the persistent objects as reusable objects where the reuse decisions and strategies are either made by, or strongly supported by, the Object Manager. In another important sense the work can be viewed as the creation of an evaluative context within which it is becoming possible to identify and promulgate some standard software development objects and operations.

The Object Manager must support environment extensibility. As noted above, environments must be able to smoothly integrate an ever-widening collection of tools as our grasp of the software development process grows. Further it seems important to see that different users are able to tailor their tool support systems to meet their needs as their sophistication grows and as the nature of their job changes. Thus it is clear that current environments must be designed to be flexible and extensible. The Object Manager can

support these requirements by managing its central data repository as a collection of typed objects in such a way that the set of types can be expanded. It must assure that each object type is created by at least one tool capability. Once the possibility of creating objects of this type has thereby been established, such objects can then be exploited as inputs to other tool capabilities, perhaps thereby resulting in the creation of objects of other new types.

The Object Manager should reuse the final or intermediate results of previously invoked tools. An important key to being able to do this is to view tool functions as being the concatenation of the actions of a number of smaller tool fragments. Under Odin, these tool fragments have no direct knowledge of, or communication with, each other. Instead, each fragment is invoked in turn by Odin. Each fragment produces as outputs data objects (usually large files or data bases of information) which are presumed to be needed by subsequent tool fragments, and are therefore retained as persistent objects. Odin names and stores these objects in a systematic way. They are then available for either immediate or deferred reuse. Each subsequent request for an object is handled by Odin in such a way as to make the best and maximal use of the objects already created and stored in its object store. The strategy used is to carefully name each object in the object store in such a way that the name accurately reflects the way in which the object either has been, or could be, derived from the most elementary objects in the store by a sequence of tool fragments. This fully elaborated name then suffices as a guide which can be used to effectively and efficiently search the object store. If the requested object is already there, the search terminates with a pointer to it. If it is not, the search terminates at an object or objects which represent(s) some progress from the store's most elementary objects towards what has been requested. Odin then uses the fully elaborated derivative name to provide a prescription of the sequence of tool fragments which must be invoked to take the already-existing object(s) and derive them into what has been requested.

The derivation relation, discussed above, is central to an effective object management strategy, being used to organize all objects into a "derivative forest". This is a structure that indicates which objects in the information store have been produced from which other objects in the store by the action of which tool fragments. The object store can contain two different classes of objects -- atomic objects and derived objects. Atomic objects are those which have entered the store either by means of a synthesis process such as text-editing or by explicit importation from an external store such as the host file system. Derived objects, on the other hand, are those which have been created as the direct result of the execution of a tool or tool fragment. The derivative forest organizes all objects in the file store into a collection of trees, such that each atomic object is the root of a tree consisting of all the objects which have been derived, directly or indirectly, from the root object by tool invocation sequences.

The derivative forest keeps families of objects stored together in a way which is conceptually clean and logical, and also in a way that makes it easy for the object manager to quickly and easily find objects which have been requested either directly or indirectly. This structure is also useful in helping manage the object store in the face of potentially disruptive changes.

When an atomic object (for example a source text object) is modified (for example by a text editor), the resulting source text object is considered to be a new object which is a version of the original object. The user may specify a name for this new version, in which case it becomes a new atomic object and the root of a (descendantless) derivative tree. If the user does not specify a name for this new version, then it automatically replaces the

object which was the original version. In this case, however, it is not safe to assume that objects which are derivatives of the original version (descendants of the original object in its derivative tree) are correct derivatives of the object which is the new root of the derivative tree.

Odin assures that the potential problem will be recognized by assigning a date and time stamp to each object under its control. Whenever a derived object is found to be older than any of the objects which are ancestors in its derivative tree, that derived object is treated with suspicion. It is tempting to suggest that all objects derived from an atomic object which has been edited simply be purged. Odin does not do this because some editing procedures result in superficial changes that do not alter some or all of the atomic object's derivatives. Odin incorporates difference analyzers that attempt to make this determination to avoid potentially costly re-derivations which might not be necessary. Here the derivative forest is an excellent vehicle for keeping together all objects which must be evaluated for re-derivation as a result of changes to their ancestors.

The shape and structure of the trees of the derivative forest are determined by the structure of the types of objects which Odin maintains. This type structure is embodied in the Derivation Graph. The Odin Derivation Graph is the structure which Odin uses in order to retain a record of which tool fragments are currently incorporated into the system, which types of objects they produce and require, and the way in which these various tool fragments can be synthesized and concatenated to effect the higher level tool functionality which users are able to request. The nodes of the Derivation Graph represent the range of possible types of objects in the object store and the edges represent ways in which objects of one type can be transformed into new objects (perhaps of different types). These transformations can be achieved in two different ways -- by casting and by derivation. Existing objects can be retyped by casting them from their present type to a new type. This is often useful when Odin determines that some existing file is the needed input to a particular tool fragment, but that the file is not of the type required by the tool fragment. Casting the object from its current type to a new type does not alter the contents of the object, but retyping it enables different tool fragments to access it.

Odin serves as an excellent vehicle for facilitating the flexibility and extensibility of toolsets which it integrates. The basis for the extensibility of Odin is the Derivation Graph just described. This graph indicates the way in which objects of any type can be built from other objects, perhaps of a variety of types. It is important to note that this Derivation Graph is accessed and maintained by the Odin command interpreter and is not accessible to the various tool fragments themselves. Further, the various tool support libraries through which the tool fragments access objects force the tool fragments to access only needed objects and isolate the tool fragments from any direct contact with other tool fragments. Thus tool fragments have no knowledge of the sequence in which they may be called, and are prevented from establishing reliance, explicit or implicit, upon other tool fragments. As a result, any tool fragment can always be replaced by another, provided that the replacement produces objects of the same types as those produced by the original, and draws upon objects created by the other tool fragments in the toolset. This effects a great deal of flexibility in upgrading or correcting existing tool fragments.

Beyond that, this architectural device also makes it relatively straightforward to integrate a new tool fragment into an existing toolset. The new tool fragment must first be characterized in terms of the object types it requires as input and produces as output. Its input object types must all be types already produced by existing tool fragments. Its output

types may be either partially or totally new. If some output object types are new, then new nodes must be inserted into the Derivation Graph to represent them. Edges connecting the input types to the output types -- new and old -- must then be inserted into the Derivation Graph as well. Once this has been done Odin has complete knowledge of when to invoke the new tool fragment, and how to optimize the creation of objects which it produces.

Odin incorporates a language by which the Derivation Graph can be specified and a processor for translating such specifications into actual Derivation Graphs. The language most resembles a production system. Specifications in this language are relatively straightforward to alter by the use of a text editor. Thus the process of altering or extending an Odin-integrated toolset entails only the editing of the Derivation Graph specification, the rerunning of the Derivation Graph processor, and the placement of the tool itself in a prespecified area of the host operating system's file store.

Experience to date indicates that this capability is indeed useful and efficient. Dozens of tools and tool fragments have been incorporated into Toolpack/IST -- some which were produced and some which were captured from host environments. New tool fragments have been incorporated into Toolpack/IST in as little as five minutes.

It seems reasonable and useful to characterize an object manager, as exemplified by Odin, as an interpreter for a language capable of programming the software development process. The primitive objects of this language are the objects central to the software development process -- source text, flowgraphs, test data, designs, etc. These objects can be aggregated into arrays and structures. The primitive operators of this language are the tool fragments' functional capabilities. In exploring this analogy further it is interesting to note that the extensibility which has just been described corresponds to the notion of extensibility in programming languages, being tantamount to the ability to construct new types and create new operators for them. The ability of the Object Manager to reuse objects corresponds to the ability of the command interpreter to optimize in the classical programming languages sense of the term.

2.4 An Ada Implementation of an Object Manager

The goal of this task is to provide an Ada version of a software environment object management system. This will entail (1) the creation of a design expressed in WIS Ada/PDL, (2) the development of a system coded in Ada, and (3) the production of a brief user's guide.

The Object Manager is to be designed so as to rest atop the lower level primitives specified by the CAIS (Common Ada Programming Support Environment (APSE) Interface Set) in an attempt to assure the maximum transportability of the design among various machines and operating systems. In addition, the implemented Object Manager is to sit atop such a set of CAIS primitives if made available.

2.5 Distributed Version of the Object Manager

This task is to create a distributed version of the Object Manager. An environment for effective support of the process of creating software systems in Ada must support the cooperative activities of large numbers of software workers. What seems to be necessary

is some scheme for distributing the object store which is maintained and some scheme for negotiation of sharing arrangements among the various workers on a software project.

2.6 Prototype Environment

This task is to suggest a set of tool fragments which, when integrated by the Object Manager, would compose a very rudimentary development environment.

The major thrust of this task is to deliver specifications of tool fragments which could be integrated in order to furnish to end users useful software development support capabilities. It is expected that these specifications would be delivered relatively early in the course of this project. These specifications would then be the basis for a search for such tool fragments, to be carried out by others.

3.0 Ada PDL

This section describes the proposed Ada PDL project. The intent of an Ada PDL will be to encourage the design as well as the initial and final development phases to be done in Ada. Associated with the PDL is a set of transformation and maintenance tools which are described in subsequent chapters in this report.

3.1 Objective

The chief objective of this project is to develop an advanced Ada PDL for use in developing WIS software. This advanced PDL will be native Ada to the maximum extent possible. Additionally, it will be based on the current WIS Ada Design Language (ADL) Standard [WIS 85], which will be extended by means of an annotation language to: (1) provide the basis for effective and rigorous testing, analysis and verification capabilities; and (2) capture the information necessary to promote the reuse of PDL design components. The PDL will facilitate expressing software designs at several levels of abstraction and so will be suitable for use at different stages in the software life cycle. The outputs of this part of the work will be a Reference Manual that succinctly and rigorously describes the syntax and semantics of the advanced PDL and an Instruction Manual describing its use.

A further objective is to develop some tool fragments that process the new PDL. These tool fragments have been chosen to provide functions common to several different useful automated tools and, once developed by this project, will be used by other Design Description and Analysis (DDA) Task Force projects to develop actual tools.

3.2 Scope

This project's intent is to enhance the WIS ADL in areas relating to reusability, testing, analysis and verification. This will be done by incorporating existing technology into the current WIS ADL rather than by developing new technology. However, it is recognized that any technology that is exploited for this purpose may require refinement and/or extension.

3.3 Background

The term Ada PDL in this section is used to denote a PDL whose syntactic and semantic features are entirely or predominantly identical to, or consistent with, the syntactic and semantic features of the Ada programming language, MIL-STD-1815A [Ada 83]. Several PDLs have already been defined based on the Ada programming language [NAC 85]. Some merely support the Ada programming language constructs used for design activities. Others extend the programming language with mechanisms for capturing some supplemental design information. Few, if any, provide the ability to describe a module's behavior, that is, its functionality and performance, in other than a procedural, algorithmic manner. Such specification is a prerequisite for testing and analyzing software effectively. In addition, there is little evidence of that current PDLs have a coherent approach for capturing information to facilitate the reusability of designs.

Over the past 10-15 years there has been a great deal of research and advanced development of techniques and tools to support the software testing, analysis and

verification processes. This experience has made it clear that the most useful and definitive techniques are those that are able to rely upon the existence of a specification of intent or requirements as the basis for effective testing and verification. Ideally the mechanism for specifying intent should be a set of annotations embedded in the text of the program design and code. These annotations are best expressed in a formal and rigorously defined language that should be thought of as an extension to the original WIS ADL.

Such an annotation language should also be designed to be an effective vehicle for supporting reuse. By enabling users to specify the intent of a body of design or code, the annotations themselves have the potential to serve as a specification of the capabilities associated with that object. The Categorization Scheme project, Section 5.0, will use the annotation language in this way to specify the reusability information that needs to be captured for different types of design components.

Alternately, and more in keeping with the spirit of Ada, a mechanism for specifying intent would be to use actual Ada code, taken from packages of implemented annotations. This proposal is discussed in Section 3.4.1.2.

Although the use of an Ada PDL is expected to offer many benefits, substantial improvements in the software development process will only be realized with the provision of automated tools that support the PDL. Indeed, this project is part of a much larger effort aimed at the creation of an environment for the effective support of Ada software development and maintenance. This proposed environment will consist of a powerful and diverse set of tools having as their goal the creation and maintenance of a large store of persistent software development objects. These objects, and their associated tools, will span the software development life cycle, starting with support for requirements and high-level architectural design, and continuing through to support for software maintenance.

The architecture of this environment is predicated upon the use of an intelligent object manager, as described in Section 2.0, to both store the persistent objects and to regulate the application of tools to create and maintain them. In addition, the environment should have an intelligent and active user interface.

The tool fragments to be developed as part of this project are to be designed and created within the context just described. They will take all inputs in the form of persistent objects drawn from a persistent object store with the help of an intelligent manager. Similarly, they will produce as output persistent objects that are to be stored in the persistent object store with the aid of the object manager. Accessing functions and primitives will be supplied that will be able to cause the object manager to store and retrieve such objects.

Wherever possible existing tools or tool fragments should be exploited in the development of the tool fragments produced by this project. Where necessary, such existing tools will be modified to be consistent with the tool fragment approach.

3.4 Approach

3.4.1 PDL Annotation

3.4.1.1 PDL Annotation Language Definition

The syntax and semantics of an annotation language capable of describing software functionality, with sufficient rigor and completeness to support effective testing, analysis

and verification of software designs and code, must be defined. The annotation language will be defined as an augmentation to the WIS ADL. It will support investigation of interactions among modules, and particular attention should be paid to concurrency issues. It will also enable capturing design decisions that have already been made without constraining the manner in which those decisions are subsequently implemented. The key aspect of this annotation language is that it will provide an assertion sublanguage for specifying intent and behavior. Such languages already exist (e.g., ANNA [LUCK 84a, 84b]) and these should be exploited, although they should first be reviewed in the context of other specification techniques (e.g., [STUC 75]) to identify any shortcomings and weaknesses and determine how these can be overcome. Alternative strategies for conducting testing, analysis and verification will be investigated so that the most suitable approaches can be identified. The advanced PDL will be suitable for use in all those life cycle phases covered by the WIS ADL Standard and allow different levels of specification so that the degree of detail and differing perspectives appropriate for each design stage can be effectively captured and specified.

A comprehensive classification of the different types of design components is proposed in Section 5.0. This classification approach is expected comprise multiple independent classification schemes. One example of a possible classification scheme based on the usage of Ada packages, would be: declaration group, operational abstraction, state machine abstract type, abstract object, etc. This classification approach will allow distinguishing between the different type of testing analysis and verification that are appropriate for each type of design component and so help to ensure that appropriate actions are take in each case.

The advanced PDL will include notations for capturing the results of performing the required testing, analysis and verification. The syntax and semantics of the existing WIS ADL will have to be refined as necessary to ensure a smooth integration with the annotation language.

The syntax and semantics of this initial annotation language will be carefully documented, along with any proposed revisions to the initial WIS ADL. This documentation will include guidelines for the use of the PDL.

3.4.1.2 PDL Annotation Code Definition

As an alternative to 3.4.1.1, the use of an annotation code, having a syntax and semantics as a subset of Ada, should be considered. The code would allow the direct inclusion of semantic specifications within the developing software, thus eliminating the need for a pre-processing of non-compilable annotation statements.

Packages of coded ANNA specification statements would be one implementation method for this annotation code. These packages would contain related annotation statements along with the associated subprograms necessary to provide a functional annotation environment. This annotation environment would allow constraints, initial values, etc. to be applied to declared types, subtypes, and objects. For example,

```
INDEX : INTEGER;     --| 0 <= INDEX <= in SIZE;
```

is an ANNA annotation statement that defines a restriction on the range of values that INDEX may take. This corresponds to the actual Ada declaration

```
INDEX : INTEGER range 0 .. SIZE;
```

The annotation asserts that INDEX must have a value in the range specified over the scope that INDEX occupies. A pre-processor will transform the annotation into a set of boolean expressions that will be inserted at the occurances of INDEX in order to provide checks against the constraint specified in the annotation. While this appears to be a clean and automatic way of specifying a constraint, it is very indirect and essentially non-"Ada-like".

A more direct way of implementing the assertion in the annotation without the actual Ada code would be through the use of subprograms of coded annotations. For example,

```
with ANNA_CODE;
declare
   INDEX : INTEGER;
   ...
begin
   ...
   ANNA_CODE.CONSTRAIN (NAME => "INDEX", VALUE => INDEX,
      LOWER_BOUND => 0,
      UPPER_BOUND => SIZE);
   ...
   INDEX := some_expression;
   ANNA_CODE.CHECK_VALUE (NAME => "INDEX", VALUE => INDEX);
   ...
end;
```

The CONSTRAIN procedure will keep track of objects and the constraints placed on those objects. The CHECK_VALUE procedure will compare the current value of the object given with the allowed values for that object. If the value violates the asserted bounds then a warning is issued. This differs from the use of ANNA in that the annotation code is compilable and executable Ada. A pre-processing step is not necessary.

The use of such an annotation code is a more direct and "Ada-like" method for providing semantic assertions. There are some limitations, however; for instance, the procedure CONSTRAIN can only be overloaded to extent of handling the basic, pre-defined Ada types. This limitation can be circumvented though through the liberal use of subtypes on the basic, pre-defined types, or possibly through the construction of a generic CONSTRAIN that could parameterize the type on which a constraint is to be made.

A PDL annotation code described above could be used in place of most simple assertions of constraint on the basic, pre-defined types and objects of those types, and also on initial value assumptions on objects. More complicated assertions involving records, subprograms, and packages are not as easily implemented. If this alternative is chosen, a comprehensive plan for the implementation of these features will be given.

3.4.2 Reusability Extension Definition

The initial annotation language extension to the WIS ADL, described above, will be extended as necessary to provide the notations required to capture needed reusability information. These information requirements are specified in Section 5.0.

4.0 Ada PDL TOOLS

This section describes the Ada PDL Tools project. These tools are designed to be used in conjunction with the Ada PDL described in Section 3.0 to enhance the effectiveness of using the PDL.

4.1 Objective

The chief objective of this particular project is to provide automated tools for the creation, transformation, documentation, and maintenance of Ada PDL designs.

4.2 Scope

This specific project involves the development of tools to support the creation, transformation, and documentation of PDL designs. These tools are to be designed and created within the context just described. Thus the tools must be designed as sequences of tool fragments, where the proposed fragments are designed to be general, and potentially reusable by other tool developers. In addition, the tool fragments are to take all inputs in the form of persistent objects drawn from a persistent object store with the help of an intelligent manager. Similarly, the tool fragments are to produce as output persistent objects that are to be stored in the persistent object store with the aid of the object manager.

This section describes the tools that are to be provided. While some of the tools will need to be developed from scratch, in many cases it will be possible to utilize existing tools or tool fragments to provide some of the required functionality. This is discussed more fully in Section 4.3.

The specific tools are:

a. Syntax-directed editor: Guided by the syntax of the PDL and the current stage in the design process to prompt for complete and correct input. (The level of detail appropriate at each design stage will be defined by the Advanced PDL project.) The default formatting for the input will be taken from the [WIS 85], but the user will be allowed to change the default values. This tool will also include program unit body stub generators.

b. Pretty-printer: Formats the design text according to predefined conventions. The default conventions will be taken from the [WIS 85].

c. Flexible documentation generator: Produces design documentation consistent with DoD-STD-2167A as the default case. The documentation produced will include reporting on the testing, analysis and verification requirements of design components (as defined by the Advanced PDL project) and the results of conducting such testing.

d. Requirements and testing tracking tool: Reports the relationship of the design to both the requirements documentation and the test plan.

e. Cross-reference tool: Supports summarizing the use of various entities in the design, showing where they are referenced. Examples of the type of reporting that is desirable include:

(1) Uses of variable names, procedure names, or other design unit entities throughout the design.

(2) Set and use analysis that ensures all variables are properly initialized before use.

(3) Locality of reference indicating the distance between declarations and each use of a design entity.

(4) Identification of the current level of detail of design components and all aspects of the design that have been left in a "to be developed" state.

f. Impact Analysis Tool: This tool will identify those parts affected by a proposed change, where the change may be required to either correct an error or enhance the capabilities of the software. It will operate in two modes:

(1) Given the requirement(s) that will change, it will identify all the affected design components and affected parts of the test plan.

(2) Given details of proposed changes to a design component(s), it will identify other affected design components and trace this back to identify the impacted requirements and parts of the test plan.

g. Regression Testing Tool: This tool will identify those parts of the design to be retested and the tests to be repeated to ensure no errors have been introduced as a consequence of changing the design. This will be based on the verification, analysis and testing requirements for each design component that are recorded with the components.

h. Producing graphic representations from PDL text: In order to ensure that up-to-date documentation of the design is continually available for maintenance activities, a tool that can produce a graphical representation of the structure of the design and show the data that is passed between design components is required. The graphical representation will be based on the approaches developed by Booch [BOOC 83] and Buhr [BUHR 84].

All tools will process the advanced WIS Ada PDL as the default PDL, but should be sufficiently flexible to permit their use with any similar Ada PDL.

4.3 Background

The use of Ada PDL's for design activities allows many of the benefits of Ada to be realized early on in the software life cycle. Consequently, a WIS Ada Design Language Standard was developed to ensure that these benefits are available in the development of WIS software. The previous section proposed that the WIS PDL be extended with an annotation capability that supports: (1) testing, analysis and verification processes; and (2) reuse of software designs and code. This annotation capability is expected to be based on a specification language such as ANNA [LUCK 84a, 84b]. While the use of such an Ada PDL is expected to provide many advantages, substantial improvements in the software development and maintenance process will only be realized with the provision of automated tools that support use of the PDL.

This project is part of a much larger effort aimed at the creation of an environment for the effective support of Ada software development and maintenance. This proposed

environment will consist of a powerful and diverse set of tools having as their goal the creation and maintenance of a large store of persistent software development objects. These objects, and their associated tools, will span the software development life cycle, starting with support for requirements and high-level architectural design, and continuing through to support for software maintenance.

4.4 Approach

4.4.1 Overall Development Strategy

Some tools, for example, the syntax-directed editor, will be useful in the development of other tools. Therefore, it would be advantageous if these tools were developed first. Similarly, some tools may share tool fragments and these common fragments should also be developed early on. Additionally, the feasibility should be assessed of exploiting existing tools or tool fragments and evaluate the cost/benefits in each case.

In order to promote consistency in the user interface, a sharp distinction will be made between tool fragments and dialogue fragments [HART 84, COUT 85]. The tool fragments will encompass the basic functionality of the system while the dialogue fragments will encompass all aspects of the software user interface including displays, prompts, input checking, error messages, and on-line help facilities. The dialogue fragments will be architecturally separated from the tool fragments.

Wherever possible existing tools or tool fragments should be exploited in the development of the tools produced as part of this project. For example, there is a set of Naval Ocean Systems Center (NOSC) tools which processes Ada source code, and the high degree of similarity between the PDL and Ada may allow using portions of these tools to provide some of the required functionality. Where necessary, such existing tools will be modified to be consistent with the tool fragment approach.

4.4.2 Ada PDLs

The characteristics of the group of Ada PDLs will be defined that will be processed by the tools. Examples will be given to show how a prospective user of the tools can determine whether his Ada PDL falls within the group.

4.4.3 Tool Development

The tools listed in Section 4.2 will be developed. In each case, the current WIS Ada PDL will be used in both the preliminary and detailed design phases, with the use of the advanced WIS Ada PDL phased in as it becomes available. All software will be implemented in the Ada programming language so as to run on any validated Ada compiler. Emphasis will be placed on the flexibility and portability of the tools. Some guidelines for increasing the portability of Ada code are given in [Ada 84].

For each tool, the following subtasks will be performed:

a. Develop and test the tool in accordance with DoD-STD- 2167. Reviews consistent with DoD-STD-2167 will be held.

b. Prepare supporting documentation showing the requirements definition, design, implementation and testing of the tool. Special attention will be given to recording: (1) all instances of where existing tools or tool fragments have been reused; and (2) the potential reusability of the developed tool fragments themselves. The portability features/limitations of the implementation will also be described.

5.0 REUSABLE COMPONENT CATEGORIZATION SCHEME TOOL

This section describes the requirements for a tool to be used to properly categorize software components. The categorization will allow the reuse of components more readily. Reuse of software is a fundamental objective of WIS and is a major reason why Ada has been chosen as the implementation language.

5.1 Objective

One of the crucial factors that will influence the successful introduction of a reuse technology is the ability to accurately categorize the function, behavior and other characteristics of a design component. Consequently, the primary objective of this project is to develop a comprehensive categorization scheme.

In addition, this section will also specify the requirements for a tool that can automate the categorization of reusable components.

5.2 Scope

The categorization scheme will both specify the particular pieces of reusability information that needs to be captured for different types of design components and guide a software developer in complete and consistent categorization of components.

The information needed to support the reuse of different types of design components must be stored as part of the design text of each component.

5.3 Background

While increased use of formal software development techniques and automated tools promise to improve both quality and productivity in the software development process, radical improvements are only likely when substantial amounts of proven software start to be reused.

Until recently, the only type of components that have been widely reused are small pieces of code, for example, FORTRAN mathematical routines. The arrival of the Ada programming language has triggered new interest in the possibilities of more extensive forms of reuse. A potentially large number of components will be submitted for entry to the WIS reusable component library. A full-scale, proven reuse technology is still a long way off, yet some initial steps can be taken that will not only help to develop the needed technology but will offer immediate payoffs in the design and construction of a reusable components library system.

5.4 Approach

An Ada generic package consists of a specification part and a body. The specification defines the package's interface to the outside world and its externally visible behavior. The body of the package contains the hidden implementation details

Two packages may have the same specification, but different bodies. For example, the same generic package that specifies a sorting routine may implement different

algorithms (i.e., exchange sort, quicksort). Each implementation may differ in terms of performance, storage requirements, or other small but potentially significant ways not addressed in the package specification.

Descriptive names and attributes are used to distinguish among implementations of Ada generic packages, and to locate the implementation that most closely matches the user's need. The classification scheme should meet certain requirements in order to be useful and flexible. Section 5.4.1 describes some of these requirements, section 5.4.2 defines some operations on attributes, section 5.4.3 describes how a relational database can be used to implement these operations, and section 5.4.4 gives some guidelines on the use of names and attributes.

5.4.1 Requirements for Classification

The attributes used to classify the generic Ada packages should meet the following requirements:

a. Expressiveness: Capability to express a wide range of information the user considers helpful in discriminating among implementations.

b. Capability to group generic packages into classes, e.g. sort packages, search packages, created by project X.

c. Capability to assign one generic package to two or more different classes, e.g. program unit A is both a sort package and created by project X.

d. Capability to assign an attribute to a program unit without having to supply corresponding attributes for other program units within the same class.

5.4.2 Operations on Attributes

The following operations are defined by [LAN 83] to create and examine attributes:

a. To define an attribute and values it may take:

```
DEFINE_ATTRIBUTE (ATTRIBUTE_NAME:IN, DOMAIN:IN);
```

b. To set a value for a given attribute:

```
SET_ATTRIBUTE (IMPLEMENTATION:IN, ATTRIBUTE_NAME:IN, VALUE:IN);
```

c. To examine a given attribute of a generic package:

```
READ_ATTRIBUTE (IMPLEMENTATION:IN, ATTRIBUTE_NAME:IN);
```

d. To examine all attributes of a generic package:

```
READ_ATTRIBUTE (IMPLEMENTATION:  IN);
```

5.4.3 Implementations of Operations

The retrieval of an implementation of a generic Ada package from a library is analogous to a relational database query. A relational database represents information in the form of tables. Rows represent objects being described and columns represent attributes of the objects. Similarly, a generic program unit can be regarded as an object, where a single

table describes the set of implementations in rows, and the attributes of the implementations are described in columns. Below is an example for a sorting abstraction:

IMPLEMENTATION	ALTERNATIVE	STABILITY	PERFORMANCE_WORST	PERFORMANCE_BEST
1	exchange_sort	stable	bad:O(n**2)	good:O(n)
2	quick_sort	instable	bad:O(n**2)	medium:O(nlog n)
3	heap_sort	instable	medium:O(nlog n)	medium:O(nlog n)

The operations on attributes can be implemented using a relational database query language.

Additionally, relational database "views" allow convenient assignment of reusable units to multiple classes without the underlying redundancy. A view can be regarded as a virtual table. It does not have an existence of its own, rather its contents are derived from one or more base tables. Initially each class may be represented by a base table, where a base table is represented in storage by a distinct stored file. Since users' needs for information evolve, so do the classes. A relational database view may be set up when a new class is defined, and its corresponding attributes may be derived from other classes in existing base tables.

5.4.4 Guidelines for Names and Attributes

There should be guidelines to control the attributes, their corresponding domains, and assigned values. Attribute names can not be globally unique across classes, since users would be forced to refer to each other in creating an attribute name. Instead, names can be automatically qualified with the name of a class, so users working with different classes can share the same attribute name without conflict. Each class could be organized around a concept, such as an abstraction (sort package, search package), a project, etc. An attribute name with a missing value must be handled with a special value, so it will still be processed by the retrieval mechanism.

Some attributes may be regarded as mandatory for each library unit such as:

a. Creation date

b. Created by

c. Compilation date

d. Narrative purpose

e. Alternative implementation, e.g. exchange_sort, quick_sort

f. Default implementation

g. Version

The first three attributes may be assigned automatically. The purpose is a narrative description of each unit, the alternative implementation would have a value that adheres to

the naming conventions, the default implementation would designate that implementation that would be selected by default if the attempted match is unsuccessful, and the version would indicate the implementation that is being improved upon and those that improve this current one.

Below are some examples of attributes that are typically useful for classifying software:

a. performance characteristics
b. size characteristics
c. precision estimates
d. levels of testing
e. quality factors (reliability, interoperability, maintainability)
f. ownership
g. algorithm description (high level in terms of user: mathematician, physicist)
h. restrictions (input, other)
i. repeatability (recursive or iterative)
j. lines of code
k. language body written in
l. parentage, heritage
m. warranty/liability/loyalty (cost, blame, responsibility)
n. compositional properties (dependencies, usage options)

6.0 REUSABLE COMPONENTS LIBRARY SYSTEM: CONTROL POLICIES

This section describes the requirements for a set of policies to govern the reuse of software components. A software component library prototype, described in the next section, will operate under these policies.

6.1 Objective

One of the crucial factors that will influence the successful introduction of a reuse technology is that of controlling the quality and availability of reusable components. Software developers must not only be provided with easy access to a library of reusable components, but must have confidence that when they select a component for reuse it will perform exactly as expected, with no errors or side-effects. Therefore, it is essential that the contents of a WIS library of reusable components be strictly controlled.

The objective of this project is to develop the necessary control policies, and procedures to implement these policies, and recommend the establishment of a suitable Control Board that will maintain control of a WIS reusable component library.

6.2 Scope

This section addresses the following:

a. Certifying reusable components for storage in the library.

b. Change control of faulty components.

c. Policy guidance for library control.

d. Organizing a central control body.

e. Automating the certification process.

6.3 Background

Increased use of formal techniques and automated tools promise to improve both quality and productivity in the software development process. However, radical improvements are only likely when substantial amounts of proven software start to be reused.

Central to the provision of a reuse capability is a library system that provides for storage and retrieval of reusable components. This library needs to provide sophisticated searching facilities that can retrieve not only matching components, but identify closest matching components and specify the parts in conflict, or synthesize individual components into composite components that provide the required functionality. Additionally, the library needs to report on the usage of components, identify those unavailable components that are frequently requested, and provide an up-to-date index of its contents.

6.4 Approach

The most critical issue is that of certification. Each component submitted for entry into the library must be subjected to a series of tests that address such properties of the component as quality, flexibility, conformance to standards, provision of necessary supporting data, etc. These tests must be applied not only to the design text of the reusable component, but also its accompanying documentation, test plan and results, code, etc. Any component that fails to meet the certification criteria will be returned to its developer for possible correction and resubmission. The properties that need to be tested must be identified and methods for measuring these properties developed.

While it is expected that errors will seldom be found in widely used components previous experience in software development indicates that some errors can be expected in new components. Users of the reusable components will be requested to provide reports on any errors they find in the components. Consequently, it is necessary to determine how such errors will be handled when identified. For example, who should be responsible for making the correction, should the component in question be "suspended" from use until it has been corrected, etc.

The final three control functions will be performed on the basis of reports periodically produced by the library system. These reports will: (1) identify needed components that are not currently available; (2) summarize the usage of all components in the library; and (3) provide a structured index of the library contents. In the first case, the list of needed components should be reviewed and, where necessary, appropriate development efforts initiated. Usage data can be used in several ways, for example, it may be desirable to remove infrequently used components. However, this data will also indicate the types of components that are most widely used and this may lead to further identification of needed components, or a better understanding of the desirable properties of reusable components that can be exploited in the certification process. Finally, while users of the library can find out if a required component is available by simply submitting their requirements, full use of the library will be better ensured by making an up-to-date index available. Mechanisms for making this information available must be established.

It is necessary to determine how these functions should be performed. This requires not only identifying the individual activities that comprise each function, but defining a set of policies that provide a framework for the operation of the library control facility as a whole. Since the policies will only provide guidelines for performing the required functions, it is also necessary to define a set of procedures that specify the step by step actions that implement the policies. A control body will be needed to institute the library control facility, the appropriate nature of such a body must be determined.

Finally, this project should include specifications for the automated support needed to facilitate the work of the library control facility.

6.4.1 Certification Process

A certification process must be developed. Certification concerns measuring various properties of a reusable component and authenticating that these properties meet a predefined set of allowable values. It must apply to all components kept in the library, both new and modified components, and all parts of a component, that is, the design text and supporting documentation, code, etc. While many properties will be measured to ascertain

a component's acceptability for inclusion in the library, others will be used to verify a component's categorization. For example, the values of such properties as resource utilization and performance may be needed when searching for a suitable component, yet are probably unsuitable for deciding acceptability. Whereas a property such as "Expected Failure Rate" will impact acceptability. Since much useful information about a component is not available when it is developed, but can only be gathered through its use, this measurement is unlikely to be a one time process. It will probably be desirable to repeat the process periodically, potentially looking at different properties each time.

Those properties must be identified that should be measured both to determine the acceptability of a component and to verify the categorization supplied by its developer. In the first case, the allowable values for each property must be determined. Methods for performing the measurements must be developed, and mechanisms for storing the result of each measurement as part of the component must be defined.

Certification must also address the question of which components are desirable in functional terms. For example, if a new component is proposed that is highly similar to an existing component, then it must be decided whether: (1) the new component is not needed; (2) the new component should replace the existing component; (3) both should be kept; or (4) the components should be combined in some way to yield a new component replacing the existing one. Rules for determining the appropriate course of action in different circumstances must be developed.

6.4.2 Component Change Control

Rules must be developed that specify how changes to a component should be controlled. Essentially, each time a component is changed, a new component is developed. Since the previous component is potentially already in use, both versions of the component must be kept and maintained independently. If unrestricted changes are permitted, the library of components would soon contain a huge amount of highly similar components, many of which may only be used once. Thus the benefits of reuse would soon be lost. Prohibiting all changes (except those to correct errors) will result in a similar problem since every time a software developer wants some slight alteration to a component he will just create a new one. Examples of two types of changes that may be permitted are: (1) when a more efficient algorithm for a function is discovered; and (2) an alternative way of performing a function which exhibits different characteristics is developed. In the first case, the original algorithm could simply be replaced by the more efficient one. The second type of change could be achieved by adding the second algorithm as an option made available through parameters. In both cases, changes should only be permitted when the updated component maintains its "downward compatibility", that is, it can be used unchanged by all the components which previously used it. This downward compatibility will eliminate the need to maintain lots of similar versions of a component.

6.4.3 Policy Generation

Policies must be defined that guide the performance of the library control functions. These policies should be based on a simplified version of the Ada Joint Program Office's (AJPO) policies for Compiler Validation and the Software Technology for Adaptable, Reliable Systems (STARS) Evaluation & Validation (E&V) group's certification

approaches. Procedures that implement these policies by identifying the particular steps to be followed and scheduling the various activities appropriately must also be developed. The policies and procedures should cover future review and refinement of both the policies and procedures themselves, as well as the activities they address.

6.4.4 Control Organization

The issues involved in setting up some central body that will constitute the primary mechanism for instituting the library control facility shall be investigated. These issues will include, but not be limited to the organization, its role, and needed authority of the control body. This will result in the development of a set of appropriate recommendations.

6.4.5 Automation of Certification Process

The automated support shall be defined that is needed to facilitate the certification process and other control functions.

A potentially large number of components will be submitted for entry to the WIS reusable component library, and each will require certification. In order to minimize the cost and time required to undertake each certification, the measurement activities need to be as fully automated as possible. Additionally, such automation will preclude any bias or discrepancies that inevitably occur when human agents perform such tasks.

Any other needed tools, for example a tool to perform downward compatibility checking, should be identified. Other existing tools (e.g., the NOSC Ada tools) may provide additional capabilities; these possibilities should be investigated and any outstanding requirements identified, in particular, whether a "harness" that packages the various tools appropriately for certification purposes should be built.

7.0 REUSABLE COMPONENTS LIBRARY SYSTEM: PROTOTYPE

This section describes the requirements for a reusable components library system. The library would contain components classified by the Categorization Scheme proposed in Section 5.0 and governed by the control policies proposed in Section 6.0.

7.1 Objective

The objective of this project is to investigate the feasibility of providing an "intelligent" library system for storing and retrieving reusable Ada PDL design components. This objective will be achieved by developing a prototype system that can be used to investigate various issues and demonstrate key ideas. The main output of the project will be the specification for a production version library system.

7.2 Scope

This section describes the scope of the work to be performed. It outlines the required functionality of the production version system and identifies those issues which the prototype system will be used to investigate. Refer to Section 7.3 for a discussion of the type of reuse strategy within which the library system will be used and additional background information concerning the nature of the Ada PDL designs that will be stored in the library.

7.3 Background

While increased use of formal software development techniques and automated tools promise to improve both quality and productivity in the software development process, radical improvements are only likely when substantial amounts of proven software start to be reused.

The type of software reuse discussed here is very different to the reuse of mathematical functions. It not only introduces new activities into the software life cycle, but requires a different way of looking at parts of software development process. This new view has two major components: the development of reusable parts, and the reuse of those parts.

Therefore the overall philosophy for developing a software system is essentially one of taking a number of building blocks and integrating them to form the new system. Hopefully many of the needed building blocks are already available as reusable components in a library of reusable components. When this is not the case, perhaps existing components can be modified to provide the missing parts, or else the parts must be developed from scratch. Where new or modified parts are developed, these can be considered for inclusion in the library. However, library components can also be developed as independent products in their own right.

Just as in developing any other kind of software system, the construction of the prototype system will be facilitated by the use of appropriate automated tools. This is particularly true in this case since the difficulty in defining the necessary knowledge base and control structures means that there is an increased need to examine the system as it

develops. The tools may range from expert system shells, logic programming languages to special purpose editors that help in modifying knowledge bases.

7.4 Approach

The library system must perform two basic activities: (1) store reusable components; and (2) retrieve these components on request. The first of these requires categorizing a component so that its description may be stored and later searched on, and linking this description to the stored software component itself. However, it is the second activity that poses the most difficulty.

The retrieval of components requires querying the software developer for his requirements for a reusable component and then searching in the library of reusable components for one that meets these requirements. So stated, the problem does not appear too difficult, potentially just a matter of searching on a given set of keywords for a match and returning the selected component. In which case, a simple pattern matching approach based on an associative tree [DEPR 83] might suffice. Indeed, this would probably be sufficient if the library were only to contain fairly primitive components such as abstract data types and their corresponding operations. But the library must be able store components providing different types of capabilities at different functional levels. In this case, not only will the identification of matches become more difficult, but the proportion of complete matches will decrease and the need to identify the "closest" match, or alternatives will increase.

Consequently the searching activity can be (roughly) broken down into the following steps:

(Step 1) Support developer in specifying his requirements for a reusable component or components.

(Step 2) Search library for a match.

(Step 3) If a single match is found, then:

Supply developer with the corresponding component.

(Step 4) If more than one match is found, then:

Determine which component is the most suitable,

Query the developer for additional constraints,

Supply selected component to the developer.

(Step 5) If no match is found, then:

Identify the composite component that most closely meets the requirements,

Determine the member components in conflict,

Search for suitable replacements,

Supply result to developer.

(Step 6) If this process fails, then:

Identify a set of components that collectively satisfy the requirements,

Supply to the developer.

(Step 7) If this process fails, then:

Identify closest matching individual or composite component.

Supply to developer with an explanation of conflicts.

These steps are stated simply. In reality, each includes a number of nontrivial subtasks. For example, consider Step 6 where different components are being combined. Here it is necessary to identify an appropriate set of individual or composite components, synthesize these into a new composite component and specify the functionality and other characteristics of this new component. In particular, Steps 5-7 would be difficult to implement using traditional algorithmic techniques. Instead they require making hypotheses and applying fairly sophisticated reasoning techniques. This type of reasoning can best be achieved through use of an expert system that replicates the actions of a human expert.

The provision of such a system has two preconditions: (1) a categorization scheme for specifying the functionality and other characteristics of reusable components; and (2) the ability to describe the relationships between the members of a composite component in a way that permits understanding the role of each individual member, and synthesizing new composite components. The specification capability must be developed within this project. This specification capability should be based on the Ada approach to package specifications and may incorporate some of the ideas from specification languages such as ANNA [LUCK 84a, 84b]. It will be developed as an extension to the advanced WIS Ada PDL (see Section 3.0).

The prototype system will focus on the retrieval of reusable components. Although this necessitates storing component descriptions, the prototype system need not provide automated categorization of components; instead, representative data can be constructed for the prototype to use. Additionally, the retrieval of a component need only result in the identification of the actual Ada PDL design, the prototype system need not be concerned with the documentation, code, test plans and results, etc. associated with a design.

It will provide the functionality described in the earlier paragraphs to an extent sufficient to allow investigation of the following issues:

a. Choice of architecture. This concerns selecting between frame-based [FIKE 85], rule-based [HAYE 85], or logic programming [GENE 85] architectures, or using some combination of these different approaches.

b. The use of multiple knowledge sources. Several different types of "experts" are required in this system to provide different functions. For example, determining the closeness of a match and synthesizing individual components into a cooperating group require potentially different knowledge bases. (Although the different parts of the system often perform common subtasks.) Blackboarding techniques may be appropriate.

c. The provision of explanation facilities that allow a user of the system to understand how the system selected a particular component: This type of feature is usually implemented as a simple trace of the reasoning path followed, or by providing justifications for the decisions taken.

d. The ability to modify the knowledge base: The contractor should investigate: (1) the ease (for the library support staff) of inserting, changing, or deleting knowledge while maintaining the integrity of the knowledge base; (2) the organization of knowledge into components that can support multiple functions; and (3) building learning capabilities into the system that will allow it to monitor its performance, identifying bad knowledge and removing or modifying them, and adding new knowledge. This is particularly pertinent in this application since reuse expertise is only now being developed and a system which can find flaws in its reasoning and arrive at new knowledge would be extremely valuable.

e. The ease of modifying the data stored about components. It will be necessary to periodically update some component characteristics, for example, the usage history.

f. The reliability of the knowledge: There may be instances when knowledge stored, or deduced, by the system is uncertain. Where necessary, techniques such as fuzzy logic, certainty factors or Bayesian logic should be investigated for overcoming this.

g. The provision of a reporting scheme that provides the following:

   (1) Reporting on the usage of components

   (2) A structured index of the components contained in the library

   (3) Identification of frequently requested components that are unavailable

Many of these issues are discussed, in the context of rule-based systems, in [HAYE 83].

While the intention is to exploit current state-of-the-practice techniques in building the prototype, some of the features listed above are quite sophisticated. Often the difficulty in providing these features will be influenced by chosen architecture. In each case, the contractor will investigate the feasibility and costs and benefits of providing the feature in the production version system and, wherever possible, use the prototype to provide practical demonstrations of how the feature should be provided.

Additionally, the prototype will be used to evaluate the suitability of the component categorization scheme and demonstrate the overall behavior of the system in searching for components.

This work will result in the development of a specification for the production version of the library system. This specification will provide detailed discussion of the required functionality of the library system and, based on the investigations performed, it will also define the appropriate architectural structure for the system giving the rational for each major design decision, and a discussion of possible alternatives with their costs and benefits where appropriate.

8.0 TEST AND EVALUATION TOOLS

This section describes the requirements for a set of tools to test and evaluate software throughout the entire life cycle. These tools will assist in producing and maintaining reliable and efficient code.

8.1 Objective

The chief objective of this particular project is to provide automated tools for the testing and analysis of Ada programs. Though primarily intended for use with standard Ada [Ada 83] and the WIS Ada PDL [WIS 85] extensions being developed as part of the Advanced PDL project (section 3.0), these tools should be sufficiently flexible to permit their use with other, similar Ada PDLs.

This project is one of several coordinated activities geared to demonstrate improved techniques and tools for designing and developing high quality and highly reliable Ada Programs.

8.2 Scope

This section provides a general overview of the tools. While some of the tools will need to be developed from scratch, in many cases it will be possible to utilize existing tools or tool fragments to provided some of the required functionality. This is discussed more fully in Section 8.3.

A set of Test and Analysis Components will be specified and developed for the WIS Foundation Technology Program. Sections 8.2.1 through 8.2.4 discuss the tools and techniques to be included in this contract at a minimum.

8.2.1 An Ada Test Harness

An Ada Test Harness shall be developed, capable of processing the full Ada language plus all WIS Standard PDL and Advanced PDL constructs [LUCK 84a, 84b]. This test harness shall have at least the following capabilities:

a. Test Command Processor capability for:

   (1) Providing online assistance in the use of the Test and Analysis Tools.

   (2) Redirecting keyboard input to allow test script files.

   (3) Redirecting all outputs to allow post execution analysis.

   (4) Invoking specific procedures or packages.

   (5) Activating and displaying assertion information.

   (6) Activating and displaying trace & debugging information.

   (7) Invoking system command processor or shell from within the test harness.

b. Test documentation preparation & management capability providing assistance in the preparation and modification of:
   (1) Test plans
   (2) Test procedures
   (3) Test reports
c. Test data preparation & management capability for creating and managing suites of test cases:
   (1) Parameterized generation of test data
   (2) Automatic stub generation for missing packages/modules
   (3) Controlled generation and execution of test scripts
   (4) Parameterized driver capability for testing standalone packages
d. Test Result Management Capability for studying the effects of individual as well as aggregate sets of tests:
   (1) Logging of test scenarios
   (2) Analysis of test coverage
   (3) Analysis of effectiveness

8.2.2 Dynamic Analysis

Both interactive and batch style reports shall be provided. Reports should be symbolically linked with the source code in the form of an annotated source listing. An interactive report preparation system shall provide for the analysis of individual test cases and sets of aggregated test cases. The user shall be able to mix and match the results from different sets of test cases in order to determine the efficiency of various sets of tests. [STUC 77, STUC 80]

Specific capabilities shall include interactive and post execution dynamic analysis of Ada programs showing:

a. Package level test coverage
b. Procedure level test coverage
c. Statement level test coverage
d. Branch level test coverage (Have all conditions, predicate states, cases, etc. been covered?)
e. Loop coverage (Has an iteration occurred? Have all exits from a loop been covered?)
f. Dynamic assertion checking
g. Frequency of execution at the procedure level
h. Frequency of execution at the statement level
i. Frequency of execution at the branch level

j. Timing (real, CPU, IO) at the package level

k. Timing (real, CPU, IO) at the procedure level

8.2.3 Test Data Generation

Working in conjunction with the Test Harness, the Test Data Generation capabilities should support the development of synthetic test data and test scripts for testing all levels of an Ada program.

Interface and declarative information from the source code should be used to automatically produce statistically synthesized parameters for calling packages and procedures.

Analysis of Ada source with embedded ANNA-like comments and assertions should yield additional test data aimed at testing statement, branch, and assertion coverage. Symbolic evaluation techniques should be included in the contractor's investigation of applicable techniques for producing a minimal set of effective test cases.

The Test Data Generation Component shall assist the testing and analysis of Ada programs at the following levels:

a. Packages

b. Procedures

c. Statements

d. Branches

8.2.4 Mutation Analysis

The purpose of mutation analysis is to stress an operational program to its breaking point. Analysis of the nature of the breakage is then used to predict the quality & reliability of the "subject program". This is achieved by intentionally introducing errors into a program and seeing how effective one's testing process is in discovering these intentional errors. The types of errors "seeded" into a subject program should be statistically representative of the normal types of errors one would expect to find in this type of program. In the process of discovering the artificially introduced errors it is not uncommon to find residual errors that have previously not been found. In fact, the ratio of artificially inseminated errors found to residual errors found gives one some feeling (predictive model) for the number of remaining residual errors.

The Mutation Analysis Component shall provide assistance not only in the creation of seeded errors, but also in the analysis of the number and type of residual errors discovered. It should also show their distribution within specific procedures and packages. The distribution of seeded errors should be shown to realistically reflect their normal historical distributions for similar classes of programs.

The Mutation Analysis Component shall support the testing and analysis of Ada programs showing at a minimum:

a. Effects of changing predicate states in conditional tests

b. Effects of changing boundary values used in predicate tests

c. Effects of changing boundary conditions on loops

d. Effects of changing the number of arguments and/or their ordering for invoking procedures

e. Effects of changing return values from procedures

f. Effects of changing boundaries on arrays and data structures

g. Effects of changing values and/or offsets used in array index calculations

h. Effects of changing values of constants

i. Effects of changing variable names

8.3 Background

The use of Ada for the design and development of new software systems promises to make a significant impact on the entire software life cycle. A WIS Ada PDL standard [WIS 85] has been developed to ensure that the benefits of systematic software development are maximized for WIS software. This PDL is currently being extended with an annotation capability that supports: (1) testing, analysis and verification processes; and (2) reuse of software designs and code. The annotation capability is expected to be based on a specification language like ANNA [LUCK 84a, 84b]. Included within the annotation capabilities will be an embedded assertion notation that will provide input to several of the testing analyzers. The syntax and semantics of this new enhanced PDL will be made available. While some early PDL tools are expected to provide noticeable advantages, maximum impact on the software development and maintenance process will only be provided when automated test and analysis tools can be linked with the PDL specifications. One of the major objectives of this project is to produce this necessary set of tools.

This project is part of a much larger effort aimed at the creation of an environment for the effective support of Ada software development and maintenance. This proposed environment will consist of a powerful and diverse set of tools having as their goal the creation and maintenance of a large store of design, program, & testing information. This information will be captured in the form of "software development objects". These objects, and their associated tools, will span the software development life cycle, beginning with requirements analysis and high-level architectural design, and continuing on through maintenance.

The architecture of this environment is based upon the use of an intelligent object manager, both to store the critical life cycle objects and to regulate the application of tools to create and maintain them. In addition, the environment will have an intelligent and active user interface, greatly aiding and easing the user's contact with these critical life cycle objects and their associated tools.

Another key architectural feature of the proposed environment is that the tools, wherever feasible, are to be built by the concatenation of smaller "tool fragments". Thus, wherever possible, contractors should design and develop small modular packages in creating the delivered tools. Capabilities from the intelligent object manager, Section 2.0, will assist in the synthesis of these tool fragments into the larger required tools.

This involves the development of tools to support the testing and analysis of Ada programs. These tools are to be designed and created within the context described above. Architecturally, the tools must be designed as sequences of tool fragments. These proposed fragments shall be designed with clean and consistent interface specifications supporting reuse by other tool developers. The tool fragments shall communicate via "messages" with the central object repository, taking all inputs in the form of "objects" drawn from the central project object store. Similarly, the tool fragments are to produce as output "objects" that are to be stored in the central project data repository with the aid of the object manager.

Accessing functions and primitives will be supplied which will allow the object manager to store and retrieve such objects. In addition, a separate and distinct interactive user interface to the proposed tools should be produced. This user interface package shall be designed and implemented as a separate tool fragment with clean interfaces (enabling the user interface to be easily replaced) to all other operational components of the test and analysis toolset. The interface should assist the user in exploring the contents of all test and analysis objects created by the test and analysis tools.

NOSC and several other government organizations are developing libraries of potentially reusable "Ada packages". Present Ada tool fragments should be investigated.

9.0 SOFTWARE METRICS ANALYSIS TOOL

This section describes the requirements for a tool to assist in collecting an analyzing various software metrics. The first part of these requirements will be to identify the measures to be taken and the second part will be the implementation of the tool.

9.1 Objective

Measurement can bring visibility to the software product and process which is essential for effective project management and control. Design metrics, for example, provide the opportunity to assess a software system's basic quality before investing in implementation. They can help to point out deficiencies within a design as well as to compare the quality of alternative designs.

The objective of this project is to provide the WIS program with:

a. A coherent framework for defining, collecting, and interpreting metric values.

b. Automated tools to minimize the overhead involved in collecting relevant information while enhancing its accuracy, consistency, and completeness.

c. Effective metric analysis tools and user interfaces to present the information in an understandable and usable format.

d. A database to maintain an historical record of metric values

An additional objective is to design the metrics collection and analysis tools so that they are easily tailorable to the needs of individual software projects, methodologies, and classes of users as well as readily extensible to allow for the identification and collection of additional metrics.

9.2 Scope

Phase I of this project involves the identification of metrics which can be collected from a design specified using the WIS Ada Design Language Standard [WIS 85] and from an implementation in the Ada programming language [Ada 83]. Phase II involves the collection of additional metrics from a design specified in an advanced PDL as well as metrics to characterize various aspects of the software process (e.g., changes and effort). This project also involves the development of tools to collect and analyze the various measures and to present the data in a form which is useful to the personnel associated with a software project. Finally, the project includes a demonstration of the resulting metrics capabilities.

A major emphasis will be on identifying metrics that meaningfully characterize a software system's architectural and detailed design. Emphasis will be given to metrics that serve a diagnostic function by pointing to the precise nature of a suspected problem, such as metrics which reflect violations of specific design principles [KELL 85], rather than those that provide only general information (such as showing that one module is "more complex" than another without leading to specific suggestions for improvement).

This project encompasses metrics taken from preliminary or architectural design, detailed design, and code. The focus of metrics for architectural design will be on the relationships between components (for example, data coupling, temporal relationships

among tasks). Metrics taken from detailed design and code will focus on characteristics of the individual components (e.g., size, complexity, cohesion, coding style).

This project is not concerned with the areas of cost modeling or system performance modeling which represent specific applications of measurement and which merit separate technology-upgrade efforts in their own right. It is, instead, specifically concerned with measuring (and improving) various quality aspects of the design and code including such characteristics as usability, reliability, maintainability, and reusability.

9.3 Background

Quantitative measures are already recorded on most large-scale projects. For example, resource expenditures are basic quantities that must be predicted, tracked, and continually updated. Size (typically measured in terms of lines of source code) is another quantity that is generally estimated and tracked. However, this information is usually forgotten once a development is completed, thereby negating any possible predictive value for future projects. Much additional information is not collected that could be useful in guiding development activities, in estimating resource requirements during the operational phase, and in providing an historical record for future projects.

An example of a useful framework for specifying and measuring software characteristics is provided by the work of McCall [MCCA 77] (and extensions to that framework found in [BOWE 83]). This framework takes the form of a hierarchy, at the highest level of which is a set of eleven software quality factors. These represent the management- and user-oriented quality concepts of reliability, usability, maintainability, reusability, portability, flexibility, interoperability, testability, efficiency, correctness, and integrity. Each quality factor is decomposable into a set of criteria which represent underlying software-oriented characteristics, such as simplicity and modularity. Each criterion, in turn, is decomposable into specific metrics. Given a low score on one or more of the quality factors (e.g., poor reusability) one can work downward through the hierarchy to locate a low score on one or more underlying criteria (e.g., poor modularity). One can then examine the individual metric values to isolate and correct specific quality deficiencies.

For the most part, the low-level metrics reflect the degree to which various design and coding principles are followed. Thus, even in the absence of extensive empirical validation showing the extent of correlation between the metrics and such measures as maintenance costs or number of errors, the metrics can serve as a useful source of guidance and feedback during development by characterizing the extent of adherence to those design and coding principles [KELL 85].

An approach such as the McCall hierarchy serves as a useful starting point because it provides a coherent framework for the multitude of possible metrics that can be used to characterize a software product. Some amount of tailoring and expansion of the framework is expected, both to reflect the proper use of Ada as a design and programming language and to include additional metrics not contained in the original framework.

One major extension could be the inclusion of visibility measures. In the Ada programming language, issues of information visibility and coupling are very important since many of the benefits claimed to result from the proper use of the language (e.g., maintainability, reliability, reusability) will be dependent on design techniques which

explicitly control the flow of information through a design, for example, through the use of Ada packages to define abstract data types and thereby hide implementation details of the data representation and associated algorithms to operate on those representations. Metrics which reflect the extent and type of information access or potential access will play an important role in evaluating this aspect of the quality of a design and in pointing to areas which are likely to be resistant to change. Examples of such metrics include those which reflect information flow through a system [HENR 81] and visibility of the information exported by packages [GANN 85]. The visibility metrics described in [GANN 85] are of interest because they are specifically concerned with Ada design issues.

9.4 Approach

The project is broken down into two phases. Phase I focuses on the identification and automation of metrics that can be taken from designs specified using the WIS Ada Design Language Standard [WIS 85] and from implementations in the Ada programming language [Ada 83]. Phase II involves the identification and automation of metrics that can be taken from designs specified using an advanced PDL (Section 3.0). Phase II may also involve the identification of process-oriented metrics (e.g., effort, changes, errors) as well as the development of mechanisms for their collection.

In order to promote reusability of tools and of data, all tools developed will be composed of tool fragments which communicate with each other through reusable intermediate objects. These intermediate objects become a vehicle for combining various lower-level fragments into higher-level tools [CLEM]. Fragments to carry out the functions of lexical analysis and parsing will be developed by a separate contractor. One of the first activities underlying the metrics project will be to review and comment upon the interface specifications to the objects created by these tool fragments. The minimal set of object types to be made available to the metrics tool fragments will include:

a. WIS Ada Design Language Standard text

b. Ada source text

c. Lexical string

d. Abstract syntax tree

e. Symbol table

f. Semantic attribute tables

In order to promote consistency in the user interface, a sharp distinction will be made between tool fragments and dialogue fragments [HART 84]. The tool fragments will encompass the basic functionality of the system while the dialogue components will encompass all aspects of the software user interface including displays, prompts, input checking, error messages, and on-line help facilities. The dialogue fragments will be architecturally separated from the tool fragments.

9.4.9 Carrying Out the Empirical Validation

The validation will involve generating operational definitions [BAIL 85] of each quality factor (for example, mean time between failures is an operational definition of

reliability while mean effort expended per change is an operational measure of maintainability), collecting these measures from actual projects, and then comparing the observed behavior with that predicted by the metrics. The output of this task will be a report which summarizes the predictive value of the metrics and suggests additions, deletions, and modifications based on the empirical results.

10.0 PROJECT MANAGEMENT TOOLS

This section describes the requirements for a set of tools to assist in the planning and to track the development of software.

10.1 Objective

Effective project management is critical to the success of any medium- to large-scale software project. There currently exists a number of tools to help with individual management activities (e.g., spreadsheets for budget calculations, electronic mail for project communications, text editors and word-processing systems for report generation, graphic packages for producing figures and charts, and cost models for resource estimation). However, there exists little in the way of a single, integrated toolset to support these diverse but highly inter-related management activities. The objective of this project is to provide an integrated set of management tools.

10.2 Scope

This project entails the development of automated support to assist project managers in:

a. Identifying project activities and their dependencies

b. Estimating and tracking resource expenditures

c. Identifying and scheduling project milestones

d. Assessing project status and likely cost to complete

e. Specifying and tracking product quality

The toolset should allow project managers to examine various tradeoffs between schedule, quality, and costs and to conduct "what-if" kinds of analyses (e.g., "What will be the effect of using more experienced designers?"). From the user's perspective, these functions will be invoked from within a text/graphics editor which is integrated with a database management system and electronic mail.

10.3 Background

10.3.1 Architectural Framework

This project is part of the WIS Foundation Technologies Program, the purpose of which is to provide an automated environment for the development, management, and support of WIS software. The environment will consist of reusable tool fragments, reusable data objects, and an object manager. An obvious example of a reusable tool fragment is a parser which forms part of a compiler as well as a syntax-directed editor, a metrics-analysis tool, and so on. Smaller, modular pieces are to be identified which are required in order to create the tools they need and, when available, to reuse fragments that have already been developed by other contractors.

Any given object, such as an Ada library unit, may undergo several transformations in the course of being processed by the tool fragments. For example, an Ada library unit

may be represented in the form of Ada source text, a compiler listing with error messages, a symbol table, a parse tree, and executable object code. These different representations form different "views" of a single object. In the same way that tool fragments can be combined in more than one way to form different tools, a given view of an object may be accessed by more than one tool or tool fragment. By storing the intermediate representations as well as the beginning and final end-products, one can achieve reuse of these object views as well. The various object views along with their access procedures represent abstract data types with the tool fragments being the users of those types. The use of abstract data types is intended to promote modifiability of the environment as well as reuse.

A specification will be provided of the logical contents of each object view along with the logical operations to store and retrieve information from that view. Once the interface to the different views of an object have been defined, work can proceed on the tool fragments which access those views.

The objects and their associated tools will span the entire software life cycle, beginning with requirements analysis and architectural design and continuing through maintenance.

10.4 Approach

This project entails the development of tools to support project management. These tools are to be designed within the context just described. Thus, the tools must be designed as sequences of tool fragments, where the proposed fragments are intended to be general and potentially reusable by other developers.

Whenever possible existing tool fragments should be exploited in the development of the tools developed as part of this project.

10.4.1 Functionality of the Tool Fragments

The general capabilities of the tool fragments are described in the following Sections.

10.4.1.1 Support for Identifying, Organizing and Tracking Project Activities

The toolset will support the project manager in identifying and tracking project milestones and activities (creating a work-breakdown structure or its equivalent). It should provide facilities for creating and updating graphical aids such as PERT and Gantt charts and should be capable of carrying out a critical-path analysis given a PERT chart or similar activity network as input. The toolset should track milestone completion down to the level of individual software components and work assignments and should inform the project manager of significant deviations (where "significant" is determined individually for each project). When a given activity falls behind schedule, information should be provided about the impact of the slippage on major project milestones. If the activity is not on the critical path, the toolset should identify activities that can be carried out in parallel. Updating of schedules should occur automatically.

10.4.1.2 Support in Estimating and Tracking Resource Expenditures

The toolset should support cost estimation for a variety of purposes including rough order-of-magnitude estimates early on in the life cycle as well as detailed estimates at the level of individual software components and work assignments later on in the development cycle. The internal workings of the model that is used as the basis for this cost estimation activity shall not be proprietary. COnstructive COst MOdel (COCOMO) [BOEH 81] is one possible candidate which is not proprietary and for which there is the advantage of Boehm's *Software Engineering Economics* text which describes the use of the model in considerable detail.

The toolset should allow for comparisons of planned versus actual expenditures, tagging significant deviations. Cost estimates should be updated at major milestones. The cost model should be tunable to individual organizations. Tuning should include not only the ability to change weights of various parameters but to add and subtract entire factors. There should also be a multiple regression capability to provide help with the tuning.

10.4.1.3 Support in Personnel Planning, Allocation, and Performance Evaluation

Given estimates of the manpower needed for various activities and software components across the different life cycle phases, the toolset should provide support in identifying the mix of personnel skills needed, in identifying shortages and excesses of personnel with particular skills within the organization and in assigning individuals to various activities. The toolset will provide ready access to Unit Development Folders or other work output useful in evaluating the performance of project personnel.

10.4.1.4 Support in Specifying and Tracking Product Quality

Costs and schedules are not the only concerns of effective project management. For example, the frequency and distribution of changes to a software component can be a useful indicator of potentially serious quality problems. The toolset will support the project manager in specifying and tracking various quality attributes.

Distribution List for IDA Paper P-1893

Sponsor

Maj. Terry Courtwright — 5 copies
WIS Joint Program Management Office
7798 Old Springfield Road
McLean, VA 22102

Maj. Sue Swift — 5 copies
Room 3E187
The Pentagon
Washington, D.C. 20301-3040

Other

Col. Joe Greene — 1 copy
STARS Joint Program Office
1211 Fern St., Room C107
Arlington, VA 22202

Defense Technical Information Center — 2 copies
Cameron Station
Alexandria, VA 22314

CSED Review Panel

Dr. Dan Alpert, Director — 1 copy
Center for Advanced Study
University of Illinois
912 W. Illinois Street
Urbana, Illinois 61801

Dr. Barry W. Boehm — 1 copy
TRW Defense Systems Group
MS 2-2304
One Space Park
Redondo Beach, CA 90278

Dr. Ruth Davis — 1 copy
The Pymatuning Group, Inc.
2000 N. 15th Street, Suite 707
Arlington, VA 22201

Dr. Larry E. Druffel — 1 copy
Software Engineering Institute
Shadyside Place
580 South Aiken Ave.
Pittsburgh, PA 15231

Dr. C.E. Hutchinson, Dean — 1 copy
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

Mr. A.J. Jordano — 1 copy
Manager, Systems & Software
Engineering Headquarters
Federal Systems Division
6600 Rockledge Dr.
Bethesda, MD 20817

Mr. Robert K. Lehto — 1 copy
Mainstay
302 Mill St.
Occoquan, VA 22125

Mr. Oliver Selfridge — 1 copy
45 Percy Road
Lexington, MA 02173

IDA

General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Ms. Karen H. Weber, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Mr. Mike Bloom, CSED	1 copy
Ms. Deborah Heystek, CSED	1 copy
Mr. Michael Kappel, CSED	1 copy
Mr. Robert Knapper, CSED	1 copy
Mr. Clyde Roby, CSED	1 copy
Mr. Bill Brykczynski, CSED	1 copy
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies